Wrocław, 9 czerwca 2024

Paradygmaty Programowania Obiektowego

Laboratorium 6

Wzorce projektowe

prowadzący: *mgr inż. Marta Lampasiak*

1 Wprowadzenie

Wzorzec projektowy

Wzorzec projektowy (ang. *design pattern*) to uniwersalne, sprawdzone w praktyce rozwiązanie często występujących, powtarzalnych problemów projektowych. Opisuje powiązania i zależności pomiędzy klasami oraz obiektami, ułatwiając tworzenie, modyfikację i utrzymanie kodu źródłowego. Warto zwrócić uwagę, że wzorzec projektowy stanowi opis rozwiązania, a nie jego implementację.

W ogólności wzorce projektowe można podzielić na trzy główne kategorie, wzorce:

- kreacyjne (ang. *Creational pattern*),
- strukturalne (ang. *Structural pattern*),
- behawioralne (ang. *Behavioral pattern*),

Wzorce kreacyjne dotyczą procesu tworzenia obiektów. Służą do ich tworzenia w elastyczny sposób, niezależnie od konkretnej implementacji. Przykładowe wzorce kreacyjne to: Fabryka, Singleton, Prototyp, Budowniczy. Na zajęciach zostanie rozpatrzony wzorzec Fabryka.

Fabryka

Fabryka jest bardzo powszechnie używanym i łatwym w zastosowaniu wzorcem projektowym. Wyróżniamy cztery rodzaje fabryk:

- **Factory** (fabryka),
- **Factory Method** (metoda wytwórcza),
- **Static Factory** (fabryka statyczna),
- **Abstract Factory** (fabryka abstrakcyjna).

Głównym celem fabryki, podobnie jak w rzeczywistym świecie, jest wytwarzanie obiektów. Dzięki Fabryce można ukryć szczegóły implementacyjne procesu tworzenia obiektów.

W celu zrozumienia praktycznego zastosowania wzorca Fabryki (w jego najprostszym wydaniu), można rozpatrzyć następujący przykład (kod został umieszczony poniżej). Pewna firma potrzebuje oprogramowania do generacji faktur dla klientów w zależności od kraju ich pochodzenia. Każdy kraj może mieć różne wymagania, takie jak format daty, kierunek ułożenia tekstu czy waluta. Obsłużenie wszystkich możliwych kombinacji za pomocą jednej klasy lub metody spowodowałoby nieczytelny i trudny do rozbudowy kod. Dlatego warto skorzystać ze wzorca projektowego.

W przedstawionym przypadku, publiczna metoda Fabryki pozwala na wygenerowanie faktury dla określonego kraju. Stworzenie prostej Fabryki, w której faktura dla każdego kraju jest tworzona w osobnej prywatnej metodzie, może być kolejnym krokiem w projektowaniu rozwiązania.

```cpp
#include <iostream>
#include <string>
using namespace std;

class Invoice {
public:
    virtual string generate() = 0;
    virtual ~Invoice() {};
};

class UsaInvoice : public Invoice {
public:
    string generate() {
        return "To jest dana faktura dla USA";
    }
};

class KoreaInvoice : public Invoice {
public:
    string generate() {
        return "To jest dana faktura dla Korei";
    }
};

class InvoiceFactory {
public:
    Invoice* create_invoice(string country)
    {
        if (country == "USA") {
            return new UsaInvoice();
        }
        else if (country == "Korea") {
            return new KoreaInvoice();
        }
        else {
            throw invalid_argument("Invalid type");
        }
    }
};

int main() {
    InvoiceFactory factory;

    Invoice* UsaInvoice = factory.create_invoice("USA");
    cout << UsaInvoice->generate() << endl;

    Invoice* KoreaInvoice = factory.create_invoice("Korea");
    cout << KoreaInvoice->generate() << endl;

    delete UsaInvoice;
    delete KoreaInvoice;

    return 0;
}
```

Metoda wytwórcza (ang.*Factory method*)

Wzorzec Metody Wytwórczej zapewnia interfejs do tworzenia obiektów, ale faktyczne tworzenie instancji obiektów pozostawia klasom pochodnym. Pozwala to na elastyczność w tworzeniu obiektów i sprzyja luźnemu powiązaniu pomiędzy wytwórcą (kodem klienta) a konkretnymi produktami.

Na ten wzorzec składa się pięć głównych komponentów:

1. **Creator (Abstract Creator)** – wytwórca abstrakcyjny (ogólny): klasa abstrakcyjna lub interfejs odpowiedzialny za deklarację metody wytwórczej. Zapewnia wspólny interfejs do tworzenia obiektów, ale tworzenie konkretnego obiektu odnosi się już do konkretnych twórców (*Concrete Creators*).

2. **Concrete Creator** – konkretny wytwórca: konkretne podklasy, które implementują zadeklarowaną w *Abstract Creator* metodę wytwórczą, zwracającą określony typ produktu. To te klasy są odpowiedzialne za tworzenie instancji konkretnych produktów. Hermetyzują one logikę tworzenia produktów.

3. **Product (Abstract Product)** – produkt abstrakcyjny (ogólny): Klasa abstrakcyjna lub interfejs obiektów tworzonych przez metodę fabryki. Definiuje wspólny interfejs, który muszą wdrożyć wszystkie produkty.

4. **Concrete Product** – konkretny produkt: konkretne implementacje interfejsu Produktu. Każdy konkretny produkt reprezentuje odrębny typ obiektu.

5. **Client** – klient: kod klienta wchodzi w interakcję z *Creator* poprzez abstrakcyjną klasę *Abstract Creator* i opiera się na metodzie wytwórczej do tworzenia instancji produktów. Wykorzystuje wzorzec metody wytwórczej do tworzenia obiektów bez konieczności znajomości konkretnej klasy tworzonych obiektów, promując elastyczność i oddzielenie klas klienta od produktu.

```cpp
#include <iostream>
#include <string>
using namespace std;

// Abstract product class
class Invoice {
public:
    virtual string generate() = 0;
    virtual ~Invoice() {};
};

// Concrete product class - USA Invoice
class UsaInvoice : public Invoice {
public:
    string generate() override {
        return "To jest dana faktura dla USA";
    }
};

// Concrete product class - Korea Invoice
class KoreaInvoice : public Invoice {
public:
    string generate() {
        return "To jest dana faktura dla Korei";
    }
};

// Abstract creator class
class InvoiceFactory {
public:
    virtual Invoice* generateInvoice() = 0;
    virtual ~InvoiceFactory() {};
};

// Concreate creator class - USA Invoice Factory
class UsaInvoiceFactory : public InvoiceFactory
{
    Invoice* generateInvoice() override {
        return new UsaInvoice();
    }
};

// Concreate creator class - USA Invoice Factory
class KoreaInvoiceFactory : public InvoiceFactory {
    Invoice* generateInvoice() override {
        return new KoreaInvoice();
    }
};
```

```
int main() {
    // Client code based on user-input
      cout << "Enter invoice type (USA or Korea): ";
      string invoiceType;
      cin >> invoiceType;

      InvoiceFactory* invoiceFactory = nullptr;
      if (invoiceType == "USA") {
          invoiceFactory = new UsaInvoiceFactory();
      } else if (invoiceType == "Korea") {
          invoiceFactory = new KoreaInvoiceFactory();
      } else {
          cout << "Invalid invoice type entered." << endl;
          return 1;
      }

      Invoice* invoice = invoiceFactory->generateInvoice();
      cout << invoice->generate();

      delete invoiceFactory;
      delete invoice;

    return 0;
}
```

Warto wspomnieć, gdzie w rzeczywistych zastosowaniach wykorzystywany jest wspomniany wzorzec:

- **Biblioteki GUI (ang. *GUI Libraries*)**: Struktury GUI często używają tego wzorca do tworzenia komponentów interfejsu użytkownika, takich jak przyciski, okna i okna dialogowe. Różne systemy operacyjne mogą mieć różne implementacje tych komponentów.
- **Tworzenie gier** (ang. *Game Development*): Podczas tworzenia gier ten wzorzec jest używany do tworzenia różnych typów obiektów, postaci lub broni w grze.

2 Zadania

Wykonując poniższe zadania możesz wyjątkowo umieszczać wszystkie klasy w pliku **.cpp*, aby usprawnić pracę (ponieważ klasy nie będą bardzo rozbudowane, natomiast pamiętaj, że w ogólności należy dzielić program na pliki źródłowe i nagłówkowe).

Zadania na ocenę 3.0

Na podstawie podanego w instrukcji przykładu, wykorzystaj wzorzec **Fabryka** i zaimplementuj tworzenie postaci w Twojej grze.

- Zdefiniuj klasę bazową `Character` z czysto wirtualną metodą `showInfo()`, która będzie wyświetlała informacje o postaci.
- Zdefiniuj klasy pochodne, które dziedziczą po `Character`, np. `Warrior`, `Mage` i `Archer` (możesz również utworzyć inne postaci – to Twoja gra, ważne, aby było ich conajmniej trzy). Każda z tych klas powinna implementować metodę `showInfo()`, aby wyświetlać tekst – unikalne informacje dla danej postaci. Sam zdecyduj, jakie to mogłyby być informacje, aby zachować sens w odniesieniu do postaci, przykładowo: "Postac lucznika - ekspert w lucznictwie i zwinnosci.", (natomiast tego przykladu już nie możesz wykorzystać ;)).
- Zaimplementuj Fabrykę `CharacterFactory`, która będzie miała metodę

  `Character* createCharacter(string type)`,

która na podstawie przekazanego typu postaci będzie tworzyła obiekt odpowiedniej klasy. Na przykład, dla typu "warrior" Fabryka stworzy obiekt klasy `Warrior`. Jeśli podany typ nie istnieje, Fabryka powinna zwrócić `nullptr`.

- W funkcji `main()`, poproś użytkownika o wprowadzenie typu postaci, którą chce stworzyć. Następnie użyj Fabryki `CharacterFactory` do utworzenia obiektu wybranej postaci. Wywołaj na tym obiekcie metodę `showInfo()`.

Zadania na ocenę 4.0

Przerób kod z poprzedniego zadania w taki sposób, aby wykorzystywał wzorzec **Metody Wytwórczej**. Opieraj się na umieszczonym w instrukcji przykładzie.

- Zacznij od przerobienia klasy `CharacterFactory`. Klasa ta ma być teraz klasą abstrakcyjną, tzw. *Abstract creator class*, a metoda `createCharacter` będzie czysto wirtualna.
- Następnie utwórz konkretne fabryki każdej postaci (*Concrete creator class*. Dziedziczą one po klasie `CharacterFactory` (dziedziczenie publiczne). Występuje w nich nadpisanie metody `createCharacter`, która powoduje utworzenie konkretnej już postaci.
- Stwórz kod klienta. W funkcji `main()`, poproś użytkownika o wprowadzenie typu postaci, którą chce stworzyć. Następnie użyj wskaźnika Fabryki `CharacterFactory`, któremu przypiszesz instancję właściwej fabryki w zależności od dokonanego wyboru. Następnie utwórz daną postać oraz wyświetl informacje o tej postaci.

Zadania na ocenę 5.0

Dodaj do klasy abstrakcyjnej `Character` pola typu `private`, reprezentujące punkty życia oraz siłę, ponieważ są to parametry każdej z postaci. Utwórz odpowiednie konstruktory we właściwych klasach, które będą nadawały wartości tym polom w celu utworzenia danych postaci. Dodatkowo uwzględnij wyświetlanie wartości pól w metodach `showInfo()`.